User guide for the CYCLONE demo

Demo description

CYCLONE is a particle system demo that uses the OpenGL-bindings available in the SGL library for Scheme. The particles are structured in regular Scheme lists to honour the Lisp (LISt Operations) fundamentals.

The demo has no interaction so it can be described as a little movie. The movie consits of a series of animations which changes the formation of the particle system. The particles are visualising a vortex of some kind, and we chose to name it cyclone.

Customization

In the code header, there are four global variables that can be used to make rather striking changes to the appearence of the movie.

Scaler

Changes to the scaler variable will affect the size of each particle. The default value is 0,02 which makes the particles small as lonely pixels.

If the value is increased, one can see that the particles shape is randomly generated for each one. This is not the usual way of working with particle systems, as the professional systems tend to use homogenous particles.

Population-size

This variable specifies how many particles that the system will consist of. This is very hardware dependent. With a powerful machine, the number of particles could be way greater than 1000, but the default is set to 200. Professional systems usually consist of tens of thousands.

Rotation-speed

As the name implies, this variable makes the particles rotate faster or slower. It is nice to set the value down to 0 in order to only watch the translations.

Cyclone-height

The cyclone can be made taller or shorter here. A shorter cyclone is denser.

Regular OpenGL setup functions are placed further down in the code. Customization concerning the colour of the particles or the light can be made there.

Data structure

There are a set of state variables contained inside the **states** object. The particles however, are located in the **population** object, which is a list. Each element in the list is a list of seven entities: the OpenGL-quad object that is the particle, x, y, and z coordinates that is the home-location of the particle and additional coordinates for the current location.

Animations

For every frame to be drawn there is a call to the **particle-caller** function. It looks at the state variable **animation-counter** and can then decide which animation to call. The available animations are:

Normal

Normal is the simplest animation. It draws each particle at their home location.

Form

Form is a line formation. All particles are located along the y-axis.

Contract

Contract makes the particle system shrink. The shrinking can be with or without affect on the y-coordinate depending on the **code** parameter.

Expand

Expand makes the particle system grow without affect on the y-coordinate.

Concave

Concave gives the particle system a quadratic look. The rotation body is recognised from integral calculus and can be described as a trumpet.

Ring-in

Ring-in does two things. First it puts the particle system into a ring formation and then it shrinks the ring.

To control the animations we use different techniques. For shrinking moments, the pythagorean theorem can be used. We calculate the distace to origo or the y-axis to get an orientation. For other moments we count the frames and use a limit.

The first animation, when the particle system is moving towards the centre of the screen, is not abstracted out as the others. The reason for this is that the same translation is applied to all particles and there is no need for making individual translations for everyone.

Our movie

The movie consists of ten parts. First, the cyclone moves into the scene from the bottom left. Then it stays at the centre and starts shrinking towards the shape of a vertical line. The third

part starts as the top of the cyclone opens up and we get a trumpet formation. The fully opened trumpet spins for a while before it closes in again.

The second half of the movie starts with the cyclone growing big. It then turns and shrinks to a line once more. The eighth part starts as the cyclone falls in towards a ring formation. The final display then shows how the ring implodes into oblivion.

Recourses

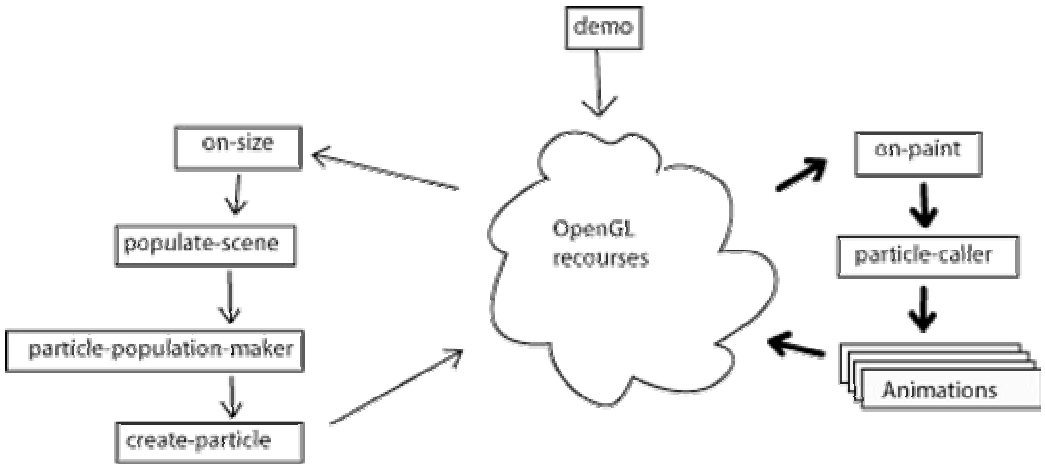
The focus in our work has been on controlling the OpenGL commands through Scheme. We therefor used logics from the sgl-example **gears.ss** that comes with the sgl package.

For particle systems theory, we used the book 3D Computer Graphics by Alan Watt.

Technical overview and HOW-TO for the future

Structure

The structure of the program looks like the box and pointer daigram:



The OpenGL abstraction cloud represents the functionality of OpenGL as well as the bindings to Scheme in the sgl-package. The left side of the OpenGL abstraction is only called once, at initiation. The right part is called every frame.

Understanding of computer graphics is required to handle the functionality provided by OpenGL. The animations we have abstracted out is the content of our graphics engine. Since it is a particle system the engine does not provide the usual functions like for example object translation. Instead the engine provides functions that work on a population.

Future HOW-TO

As always when working with graphics, the power of the computer becomes a bottleneck. We recommend that a focus is determined early that specifies whether logic or optimization is the goal. We looked at the former.

If optimization is the focal point, then the use of list data structures is out of the question. Arrays (in scheme: vectors) should be the structure to use. The slowness of lists is extra painful in our case because we have a particle system.

One trick that could come in handy is to turn off the vertical synch on the graphics card. Vertical synch is a function that synchronises the framerate of the program with the one in the monitor screen (refresh rate). When this is turned off, the program speed up dramatically.